1998

Unit test modeling: A new approach in object-oriented unit testing.

Peng. Luo
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Unit Test Modeling
A New Approach in Object Oriented Unit Testing

by

Peng Luo

A Thesis
Submitted to the College of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
at the University of Windsor

Windsor, Ontario
September 14, 1997

Abstract

In this thesis, a new approach named 'unit testing modeling' for object-oriented testing has been developed. This approach aims at addressing the issues in object-oriented testing, especially for unit testing. A set of new symbols and definitions have been developed to capture the unit structure in object-oriented system. Rules and algorithms have been developed to guide test case generation. New methods are proposed to represent various relationships in object-oriented systems. The applicability of the new approach has been discussed for different type of classes. A modeling framework has been setup.

Acknowledgments

I would like to express my sincere gratitude first and foremost to my research advisor Dr. Liwu Li for his support and guidance throughout my study at the University of Windsor. I am grateful to Dr. Indra A. Tjandra for his thoughtful comments for the work. I would like to thank Dr. Purna N. Kaloni for serving on my thesis committee. I also would like to thank Dr. Richard A. Frost to be my committee chair. I also appreciate other professors and staff in computer science department for their support and help. Last, but not the least, my most sincere gratitude must go to my wife for her understanding and loving support so that I could reach my goal.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing is not new for software development. It comes with software development procedure naturally. Recently more attention has been paid to software testing. In modern society, various kinds of software play an important part in social life, advanced research, industrial application, education and so on. In most of these situations, people cannot afford the failure of software. As software systems are becoming larger and more complex, testing is becoming more important than before. Software testing is difficult problem coming with the development process. The discussion about the time that is spent on testing and maintenance of some software systems is given in [16], [17].

Software is different from other engineering products in terms of testing. At the very beginning of the software development history, there were several definitions of software testing [1], [9], [18]. Software testing is difficult to perform. First, software testing is not well understood. As Henderson stated in [8]"There is a property of software which makes computer programming unique among the engineering discipline. The material from which the artifacts are constructed is of a mathematical kind. The material is not found in nature and governed by physical laws, but is an artificial creation which obeys laws which we ourselves control". From this point of

view, the difficulty of software testing arises from the man-decided rules which may be unpredictable. Second, the testing process is not well understood. In such a situation, some parts of the software may be constructed from verified derivation, but the majority involves some human input, some notion of creativity or intuition. The different talents and abilities of individual programmer and variety of environmental and personal factors might affect the work. Therefore, it is argued that the processes involved in creating software cannot be fully formalized, analyzed and understood. The detailed discussion on this issue can be found in Shaw [26]. Figure 1 lists the semantics of testing [23].

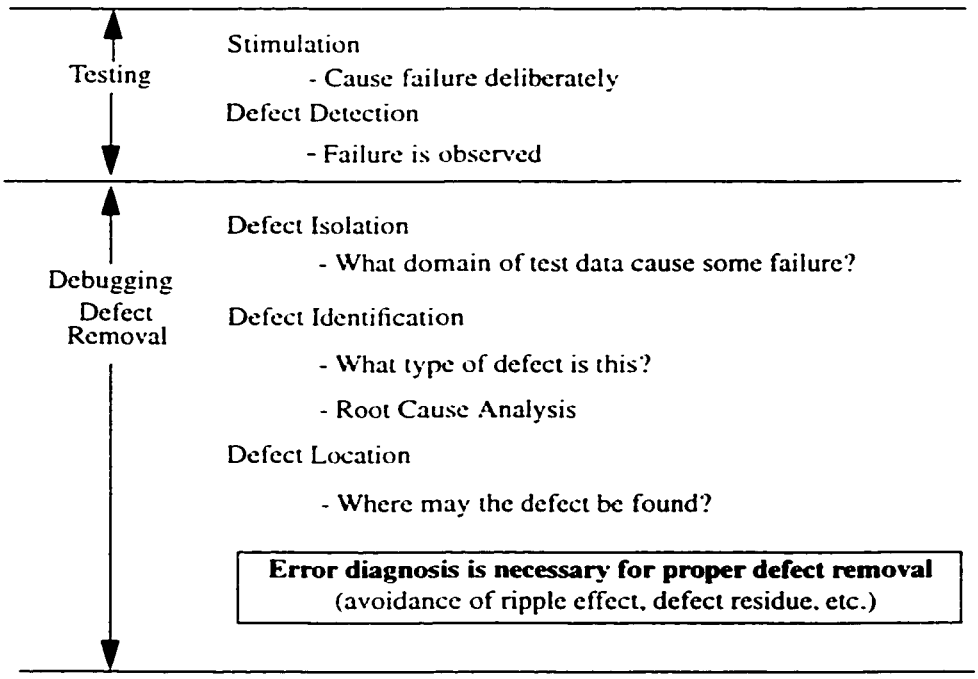

Figure 1 Semantics of testing

In the software testing community, it is well known that owing to the complexity of the programs, we cannot show programs to be correct by testing them. In most of the situations, the resource requirement does not allow us to perform full testing. For example, given a very simple program, exhaust testing cannot try all the possible inputs in a practical way simply because of the resource limit. The problem of exhaust testing is classified as being *intractable*, which means depending on a computer is difficult because of the inordinate demand for resource. Even if the input domain was small enough for exhaust

testing, another problem still exists. There may be test cases for which the program does not halt. It is therefore necessary to identify these cases so as to be able to complete the testing. The problem of identifying non-halting cases is classified as being *undecidable*. When the problem domain grows, the test data set grows too. Therefore testing can choose only a subset of the input domain in most applications. The size of the subset is determined by the time available to do the testing and by the techniques employed. Testers want to choose subsets that give the greatest confidence in the program under test. Software testing aims at finding methods and developing algorithms with which we can exercise a program as much as possible and in the meantime the testing data sets are minimized to meet the resource requirement.

Some methods and notations have been developed for inspecting the internal structure of source program. Directed graph representation is a widely used method. A directed graph consists of nodes connected by arcs (or edges) with arrows to indicate direction. The nodes represent blocks of statements and the edges indicate precedence. With the directed graphic representation, some frequently used concepts such as *path* and *branch* can be easily represented. Using these notations, we can easily find that even a very small program with a few loops and branches may produce many paths which will cause intractable problem. In Weyuker's paper [31], it was shown that there is no algorithm that can determine whether or not a given statement, branch or path in a program may be exercised, or whether or not every such unit may be exercised.

There are different approaches to perform software testing. The bottom-up approach means testing the lower level components first. The disadvantage of bottom-up approach is that we cannot find the system design errors early. The lowest-level components such as procedures and functions are usually the first executable items to be produced. They should be tested by the designers using the structural (white-box) and functional (black-box) techniques. At this point, the subject under test is relatively small and can be subject to a comprehensive testing.

The next step is to test groups of functions and procedures in modules. Test cases are derived from the specification. Following this comes the integration testing stage, in

which all the modules are linked together. The final stage is system testing, which tests the functionality of the whole system.

The early testing techniques were preoccupied by achieving and defining higher level of coverage based on the structure of the program and the way it uses data. In the last ten years, there has been a growing interest in the techniques of fault-based testing which, instead of trying to exercise the program with more and more data, have concentrated on generating data that reveal likely faults in a program [5], [6], [14], [21], [32]. It is this kind of technique that could become very powerful in the future.

Object-oriented design and programming [13], [20], [28] is relatively new comparing with the traditional structured programming paradigm. Object-oriented techniques introduce new concepts such as inheritance, encapsulation, etc. into the software industry. The concepts are important for the quality, flexibility and robustness of software. While these concepts enhance the productivity of software, they also introduce new issues for software testing. Although object-oriented software testing has its own properties, it still shares some common concepts in traditional testing method. Before we go into detailed discussion about object-oriented testing, some useful testing concepts are discussed in section 2 because they apply to object-oriented testing too.

Chapter2

Testing Review

2.1 Testing and software development cycle

Testing tasks occur during the various phases of software life cycle. The most commonly used model of the software development is the waterfall model [24], which includes several stages, see figure 2. The model is suitable for structured program development. The major development activities that take place during the requirements analysis phase are the elicitation and clarification of requirements. The major testing activity that occurs during this stage is derivation and verification of requirements. In the latter stages of the projects, the requirements are converted into the system tests, which determine whether a system meets the user's requirements. Another model is the spiral model which is employed by object-oriented design [2]. In the spiral model, a cycle contains several steps including requirement analysis and refinement, quick design and fast prototyping, customer evaluation of prototype, refine prototype. A product development process is a interative process which repeats the cycle until the final engineering product has been done.

The requirements are high level abstractions. As the software project proceeds, they will be expanded into a number of tests. The expansion of the requirements usually takes place during the sys-

tem design and implementation step. Keeping a relatively high level system specification will make it easy to present the design to the customers. Although the requirements can be established in the final stage of the system design, it is important that they are established as early as possible so that the developers who are in charge of the requirement analysis can check the quality and test the correctness of requirement specification earlier. The test plan should be developed in the requirements analysis stage although only an outline is needed. A test plan should be a very bulky document, which usually contains the following aspects [12]:

- *The organizational responsibilities for the various tasks in the testing program*
- *The methods that are to be used to review documents associated with the test process*
- *How the outcome of tests will be checked and monitored, and how discrepancies associated with tests will act on*
- *A description of the categories and levels of testing that will occur during the project*
- *The list of the tests that are to be carried out, together with the expected time that they will be executed*
- *The various hardware configurations that are to be used when a test is executed*
- *A description of each test that is to be carried out*
- *A list of verification requirements*

The information should be abstract. It is important to create this document in the requirements analysis stage.

During the system design, there are many testing activities. Obviously, from the first aspect in the test plan mentioned above, the verification requirements will be expanded out so that they correspond more closely to individual tests.

Unit testing is carried out by the designer or the design group who produced the unit because the designers have the best knowledge of the detailed design and implementation.

For object-oriented programming, class testing is unit testing. Unit testing corresponds to the final implementation stage.

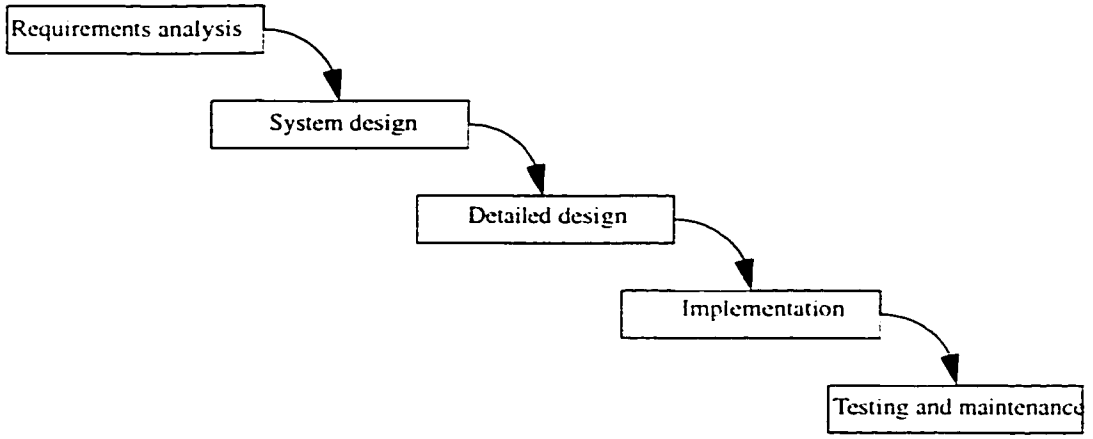

Figure 2 Waterfall model for testing

2.2 Testing terminologies

2.2.1 Different types of testing

In the history of software testing, some classification has been made. High level definitions of software testing have been developed. They can be applied to both conventional and object-oriented testing methods. Regarding the testing procedure, there are static and dynamic testing techniques. The following are the definitions of static and dynamic testing from[22].

> *static testing: static testing examines the software without executing it and encompasses activities such as inspection, symbolic execution and verification.*

> *dynamic testing: dynamic testing examines the software by generating test data for the execution of the software.*

Other concepts include the black-box testing[10] and white-box testing[19]. These notions are related to the program's structure. The following definitions are also from[22].

black-box testing: *black-box testing uses test cases that are derived without reference to the structure of the program. In another word, the software is treated as a black box and its functionality is tested with different combinations of input. It is also termed as 'functional' or 'specification-based' testing*

white-box testing: *white-box testing uses test cases that are derived by an examination of the structure of the program. It deals with the internal structure of a program. It is also termed as 'structural' or 'code-based' testing.*

Black-box testing is used to check if the software conforms to its specification. Black-box techniques have drawbacks. Without examining the code, programmers do not know how much of a program has been tested. If software behaves differently from the specification, we need white-box testing. White-box testing gives testers the opportunity to inspect the program and confidence that they have achieved some kind of test coverage, such as execution of every statement. White-box techniques are not sufficient on its own. For example, the software under examination may not perform one of its desired tasks, even the function to do this may be missing. White-box testing of the code is unlikely to reveal this. Therefore, the black-box testing is needed to point out such missing functionality. Now we can see that we should apply both kinds of testing methods to achieve higher coverage. Although classes in object-oriented systems are good candidates for black-box testing, it is also important to perform white-box testing for the units.

2.2.2 Validation, verification, and certification

This section compares of some testing related concepts. The concepts include *validation*, *verification*, and *certification*. Having internal relationships, the concepts are different. All the concepts are part of the quality engineering concepts.

Validation is the procedure that proves the product of each development phase containing the features prescribed by its requirement specification. It checks whether the end product of the current phase performs and functions as required. It is conducted continuously throughout each phase of the software development cycle. Validation of products throughout the life cycle phases needs a traceable system. Validation activities involve developers and testers.

For object-oriented design, at each iteration of the object-oriented process, the designer should ensure what he/she has produced meets the corresponding requirements. To achieve the target, the designer compares the requirement with what he/she obtained to validate the object-oriented analysis and design. In the meantime, the description of requirements will be detailed enough so that test cases can be derived from it. The strategy has the advantage of being able to trace tests back to requirements.

Verification proves that each step of the development process correctly addresses and implements the requirements. It is concerned with whether the designer builds the system correctly. It will be conducted continuously throughout the software development cycle. Like validation, verification of life cycle products needs a traceable system. Verification activities involve developers and testers.

Certification is the process of integration, verification, validation and testing of executable code throughout the development life cycle. Integration assembles software components into more complex components and systems. It is a formal, planned and managed activity. Certification is an overall process which is integrated with the software development process. It requires system traceability to track software development, integration, verification, validation activities. Certification could include other functions such as QA (Quality Assurance), configuration control, reliability, etc.

Chapter 3

Object-oriented Development

3.1 Introduction

The software engineering has undergone a serial of stages. At the early stage, the main stream was the procedural programming paradigm. The focus of the procedural programming paradigm is the processing and the algorithm. Given an application, designer decides the algorithms, procedures, functions, etc. to perform the desired task. By passing parameters into a function and outputting the result from the function, a process is done.

Now applications are becoming larger and larger and the software systems are becoming more complex. The procedural paradigm cannot handle applications well because it lacks the ability to organize big amount of data. There is a transformation from the procedural paradigm to modular programming paradigm. Modular paradigm focuses on the data organization but not on procedures. It supports the concept of data abstraction, which leads to better structure organization. The modular programming paradigm has problems. For example, although users can define types for their application, the types are not normally managed by the programming environment and compiler. Moreover, by the nature of data abstraction, a data type defined by a user behaves like a fixed black

box to perform certain defined task. This is not flexible. With a little change of functionality, the user has to defined a new type from the scratch.

It is a great benefit if a programming paradigm allows a user to capture the common property of the types and provides reuse mechanism. In the meantime data abstraction is supported. Object-oriented programming paradigm has been developed for such need.

Object-oriented programming paradigm organizes software as a collection of objects, which are implemented by *classes*. According to James Rumbaugh's definition, "Classification means that objects with the same data structure(attributes) and behavior(operations) are grouped into a class" and "a class is an abstraction that describes properties important to an application and ignore the rest".

With object-oriented programming paradigm, modeling and design can be done in different ways. Object-oriented approaches include Rambaugh's approach, Jacobson's approach, and other approaches. All the approaches try to provide formal, efficient terminologies and definitions to the users in object-oriented design. Among the approaches, Rambaugh's approach gains the popularity because of its completeness and formality. Recently, more effort has been forcused on a unified methodology, which is called Unified Modeling Language(UML). UML is an approach combined from the existing approaches. Because of the popularity and advantage of Rambaugh's approach, we will use this approach in the following discussion. The new developed method has been designed to be compatible with Rambaugh's approach.

Rambaugh's approach, also called OMT(Object Modeling Technique), includes three models:

- ***Object model*** captures the static structures and relationships of the objects in a system.
- ***Dynamic model*** captures the change and control aspects of a system.
- ***Functional model*** represents the computation of the system. It contains a data flow diagram.

Among the three models, object model is the most important one. The dynamic model relies on the concept of state machine. In the thesis, we mainly consider the object model. Other models can be borrowed from other modeling methods and can be completed in the future work. OMT object model has graphical terminations for the key concepts like *class, object, association,* etc. The newly developed object-oriented unit testing method follows these pattern but the components and relationships are reexamined in testing perspective. In the next section, some of the Rambaugh's notations are used for the discussion about the concepts of object-oriented approach. The details of OMT can be found in the following sections and Rambough's book[25].

3.2 Object-oriented concepts

Object-oriented programming paradigm supports data abstraction and software reuse. It introduces useful mechanism to enhance software design. Object-oriented development is now getting more popularity because of its powerful features. The features include *inheritance, polymorphism* and *encapsulation* etc.. They have impact on the overall software structure and the design procedure of software. Their impact on software testing is one of the most active research area in object-oriented community.

3.2.1 Classification

The concept of *classification* is the fundamental concept of object-oriented approach. In object-oriented design, objects that have the same data structure and the same operations are grouped together as the instances of a *class*. A class models a complete concept in application. It supports the concept of information hiding. The concept of class also helps eliminate the usage of global variables. A class definition consists of two parts: the *attributes* and the *member functions*. The attributes represent the state and the information of an object, the member functions are the means to change the value of these attributes. The member functions are the interface of an object to the external world. In object-oriented approach, classes are the units in the design. An application is based on classes. The instances of classes communicate with each other by sending messages. Unit testing in

object-oriented software focuses on class testing. In OMT, a class is represented by a rectangle and an object is represented by the round-corner rectangle. An example is shown in Figure 3.

Figure 3 OMT representation of a class and an object

3.2.2 Inheritance

Inheritance means the ability of inheriting the attributes and member functions from a superclass. It is one of the most powerful features of object-oriented approach. It is the engine for software reuse in object-orientation. Modern software is usually a large software system. It is usually the case that designing everything from the scratch is impossible. Using object-oriented approach, a designer can design the superclass for common properties, which a subclass can inherit from. Single inheritance means a subclass can inherit from only one superclass; multiple inheritance means a subclass can inherit from more than one superclass. Multiple inheritance increases the flexibility. It also makes the software structure more complex. Inheritance introduces interesting issues in object-oriented testing. Inheritance makes code shorter. But more extensive testing may be needed. Sometimes, inherited operations need to be retested because the operations may be invalid in the new context. For example, the descendant modifies instance variables for which the inherited operation assumes certain values. Object that overrides inherited methods and attributes must be retested. The following figure contains an inheritance relationship expressed in OMT notation.

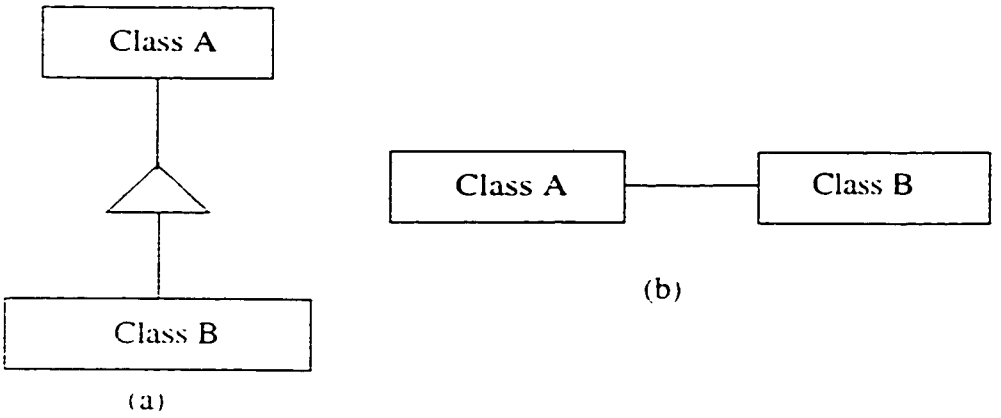

Figure 4 Inheritance(a) and association(b) relationship in OMT

3.2.3 Polymorphism

Polymorphism means the ability of taking more than one form. There are different kinds of polymorphism relationships. For exampl, template class supports horizontal polymorphism. The ovrriden functions is an example of the vertical polymorphism. This feature is useful in applications. For example, assume class *Shape* is the superclass of subclasses *Triangle* and *Rectangle*. The three classes implement the *print* function differently. When a user calls the *print* function for an object, the system takes the right one according to the object's class. This makes the program cleaner and more consistent. On the other hand, it is also an overhead for object-oriented paradigm. Because dynamic binding is associated with polymorphism, the run time efficiency is a question. Moreover, polymorphism makes testing more difficult than before. For a tester, it is not easy to trace which implementation is being tested at the moment when there are different implementations. The polymorphism features may not be known until run-time (dynamic binding), which makes white-box testing difficult. Polymorphism allows objects to be instantiated for a variety of data types. As result, we have to test all the possible forms that a polymorphism component may take. In other words, we can not extend the result of testing over one data type to another. The only solution to this is to test over the entire range of possible data types. Polymorphism complicates object-oriented testing significantly. In essence, a tester may

not know exactly which classes and which messages are being used. Polymorphism hides the complexity of the code, which makes testing more complex.

3.2.4 Association

Association relationship represents relations among classes. Just as mentioned above, an object-oriented system is based on classes. An association relationship captures a relation of classes. There are different kinds of association relationships. An association could be one-to-one or one-to-many. It also can be bidirectional or single direction. In the thesis, association relationship is expanded into class, which menas we also inspect how the components in a class associate with each other. Notations are defined to capture the relationship among the member functions and the data members. Different notations are developed for the components in a class.

3.3 Object-oriented testing

Object-oriented approach is a relatively new approach for the software development. There is work need to be done in object-oriented testing. Because of the features that are introduced by object-orientation, existing methods need to be reinvestigated and also new methods specific for object oriented features are required to meet the needs of object-oriented testing.

From the survey of the current research on object-oriented testing. The following object-oriented testing is being investigated.

1. Object testing

An object is the combination of data and operation, the starting point in object-oriented testing is testing an object, which includes:

- *testing a single operation*
- *testing an object*
- *testing sets of objects*

2. Class and cluster testing

A cluster is a group of related classes. Cluster testing tests the design of classes. The consideration is on a higher level of object testing. Class testing verifies the completeness and correctness of the class in the view of its functional and structural requirement. Class is generally viewed from an atomic, aggregate or abstract level. The topic of generic class testing addresses testing issues common to a group of classes. Functional test requirements can be derived from the use cases(different scenarios) and analysis of object specifications.

Some of the testing proposals have been made to test a generic class. For example, a suggested method is to test a test class instead of a generic class. The test class is constructed as simple as possible. Based on the result of testing to the test class, a class that derived from the generic class can be tested. The methods of testing a generic or concrete class can be found in [3], [4], [15], [7], [27], [29].

3. Integration testing

In integration testing, a tester focuses on the interaction of classes. It emphasizes the higher level performance of the design. The prerequisite is the class and/or cluster testing.

4. Automated testing for object-oriented software

Testing automation is important for large projects. Object-oriented development supports software reuse. It can be used for large projects. Automated testing for object-oriented software is necessary for large projects.

5. Design for testability in object-oriented system

Like the hardware design, design for testability is becoming more important than before. It improves the product quality and reduce the cost of testing, which weights more than 50% of the cost of the product.

The above is only a brief review of the current work on object-oriented testing. The thesis is about unit testing in object-orientation. In the following chapters, the research work on unit testing for sequencial program in object-orientation is discussed. These research results reflect the state-of-the-art works on unit testing in these area.

Chapter4

Related Research

4.1 Unit testing

In object-oriented programming, object is the basic construct. Class is the template of objects. Unit testing in object-oriented programming is to test a class or an object. Since the concept of class is not available for the procedural programming paradigm. It introduces concerns in unit testing. Some researchers have been working on unit testing for object-oriented programming. For example, state based method[30], incremental testing method which focuses on reuse the test cases for testing the inheritance relationship[7], and so on.

Comparing with the works on testing methods for structured software, research in the field of object-oriented testing has been done much less. Testing object-oriented software differs from testing structured software because of the idiosyncratic structure of the object-oriented software. Object-oriented implementations present a number of interesting issues for the tester. From the first glance, the object-oriented testing should be easier than the conventional testing for the following reasons. First, tying of a number of methods to a data item tends to make the testing simpler. This is because the procedures will act only on the data item and communicate with the outside using parameters. Therefore, the scope of side-effects is

reduced. Second, the procedures tend to be simple, performing very distinct functions. This means that the application of any of the testing methods described earlier is relatively straightforward. On top of this, using inheritance and introducing methods in subclass help users to build and test the systems in a controlled fashion. In an interesting discussion of testing, Perry and Kaiser[21] showed that object-oriented testing is not as simple as described above. They argued that the inheritance feature hides dependencies. A tester may need more time to test the subclasses. In the following sections, Perry and Kaiser's result and other object-oriented unit testing methods are briefly reviewed.

4.2 Adequate testing

Before discussing further about the object-oriented unit testing, we need to mention Perry and Kaiser's work[21] on adequate testing for object-oriented programs. They have developed test data adequacy and considered adequacy criteria in the light of the axioms. The work is on a very high level, which applies to different levels of object-oriented testing. The four axioms that have impact on object-oriented paradigm are listed below.

- **Antiextensionality**

  There are programs P and Q such that $P \equiv Q$ (P is equivalent to Q), test set T is adequate for P, but T is not adequate for Q.

- **General Multiple Change**

  There are programs P and Q which are of the same shape, and a test set T such that T is adequate for P, but T is not adequate for Q.

- **Antidecomposition**

  There exists a program P and a component Q such that a test set T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q.

- **Anticomposition**

There exist programs P and Q and a test set T such that T is adequate for P, and a set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for the composition of P and Q.

In their article, they proved that applying test adequacy axioms to certain major features like encapsulation, overriding and multiple inheritance in object-oriented programs will pose various difficulties for testing a program. When we consider new testing method for object-oriented program, we can use these axioms as guideline to make up the test criteria.

4.3 Testing on inheritance

Testing inheritance relationship is important and interesting for object-oriented testing. Some researchers have carried on in this area. In this section, I will mention the most advanced progress in this area.

Harrold and McGregor[7] have developed a testing method which is known as incremental testing. The method can be used to reduce the amount of testing needed by exploiting inheritance relationship among classes.

When using this method, a tester first tests the base classes having no parents by designing a test suite, which tests each member function individually and also tests the interactions among the member functions. Base on the result of testing parent or base class, the tester then designs the test suite for subclasses. The method uses an algorithm incrementally updating the history of the superclass to reflect both the modified, inherited attributes and the subclass' newly defined attributes. Only those new attributes or affected inherited features are tested. The effort is to reuse the test suit of the superclass. In their work, the inherited attributes are retested in the new context in the subclass by testing their interactions with the subclass's newly define attributes.

Attributes of a class are classified according to their nature of the interaction with other attributes. The attributes are classified as:

- *New attribute*
- *Inherited attribute*
- *Redefined attribute*
- *Virtual-new attribute*
- *Virtual-inherited attribute*
- *Virtual-redefined attribute*

Based on the study on the classes and the definition of the classification of the attributes. Harrold and McGregor argued that any inheritance hierarchy can be decomposed into a set of partially ordered pairs of classes. This decomposition permits them to consider only a class definition and its immediate superclass to constrain the definition of that class. They have developed an algorithm to update a testing history so that when a subclass is defined, the algorithm can be used to derived the testing history for the subclass.

They claimed that the advantage of their testing method was that completely testing a subclass could be avoided since the testing history of its superclass was reused for the design of the test suite for the subclass.

4.4 State testing for objects

State based testing is another kind of testing technique for object-oriented programming. The technique has been used for the traditional structured programming. There are several papers reporting the use of this technique on object-oriented programs. The early work on this issue can be found in Turner and Robson's paper[30]. As a complimentary technique to the functional and structured approaches, they described a technique known as state based testing, in which the test cases were generated based on the transition of object states. Their discussion was based on the traditional state testing method.

4.5 Applying traditional method

Like traditional testing methods, there are two approaches in object-oriented testing. Functional or black-box testing exercises the external interface of a class and does not consider its internal structure. Test cases in this category focus on external behavior and are derived from the functional specification of the class. It is obvious that classes are suitable for the black-box testing. This is because class is designed as black box for the user who does not need to know the implementation. In addition, white-box testing is also apply to class testing. Structural or white-box testing focuses on the internal logic of a class. Implementation details are visible and used to build test cases. White-box testing can not be used as a stand-alone method because the missing function cannot be found by this method. Using of white-box testing is most appropriate at the completion of black box testing.

Traditional testing methods can be applied to object-oriented unit testing with modification and enhancement. It is obvious that new enhancement is needed for the features like inheritance. As this kind of methods are based on the traditional testing methods. We regard them as an extension of the traditional methods.

Chapter 5

Unit testing modeling

For bottom-up testing approach, unit testing is the first step in the process. In the case of a large software project, there are several steps in software testing. Unit testing must be done before integration testing. The purpose of unit testing is to test that the unit has been designed and implemented as required. In object-oriented testing, unit testing corresponds to class testing. Object-oriented approach has introduced constrains in a class. For example in C++, there is a mechanism for access control, such as *protected* and *public* definition. Because of complex relationships and complex structure of a class, it is beneficial that a designer uses tool to improve the accuracy and effectiveness of unit testing. Overall, when a designer prepares unit test cases from the design, a unit testing model is helpful. The description of unit testing model is as follows.

Unit testing model is a model that captures the structure of a unit and internal relationships in the unit. It can be used as a guideline for software testing. The unit testing model contains a set of definitions and algorithms, which will be discussed in the following sections.

Without this kind of model, the test cases of unit testing may be ambiguous, incomplete, and inconsistent. Also the developer needs

a kind of guideline to improve the efficiency and quality of test cases. The unit testing model provides a means to capture the unit design structure and the relationships among the components in the structure. Based on this model, test cases are relatively easy to produce.

In the thesis, unit testing model includes two aspects. The first aspect is about the structure in a unit. The model captures the static structure of a design unit. It can be seen as the compensation of Rumbaugh's object model and other kinds of object-oriented model. In Rumbaugh's approach, an object model is used to capture the static structure of a system. Class is at the lowest level in the model. Our model focuses on the structure of the class. Different from the Rumbaugh's approach, unit test model deals with the internal structure of a class. The representation and relationship of the components in a class can be captured in a unit testing model. The method presents the structure similar to the cause-effect graph does. The details of the method is discussed in the next section.

The second aspect of the model is the control aspect in the model. The model focuses on the control part in the design unit. In object-oriented programming, a class is usually considered as design unit. In this case, the control aspect captures the relationships of the methods and data encapsulated in a class. This control aspect exposes how the member functions work on the data to change the class states.

5.1 Cause-effect graph and unit testing model

Cause-effect graph is a traditional testing method used for the structured programs. It is based on verification method for hardware design. In combination circuit design, a circuit is represented by a logic net of logic gates. The output of the circuit can be deduced from the combination of the inputs. To test the circuit, the input can be viewed as the "causes" and the output can be viewed as "effects". A simple example is shown in figure 5.

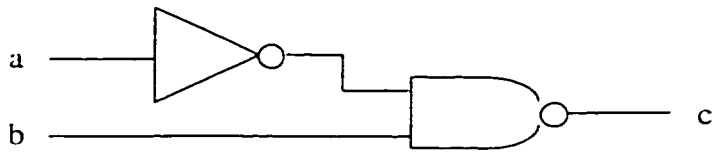

Figure 5 Causes and effect in combination circuit

In software engineering, cause-effect graph is used in software testing and analysis of requirement. First, the specification is analyzed and divided into different groups by functionality. By analyzing the input conditions, we get a set of *causes* and the corresponding results which are *effects*. The cause-effect graph is used to present the relationships between causes and effects. The method provides a means for capturing the relationships from the specification. Some constrains can be applied to the cause-effect graph to represent more complex relationships.

The commonly used operations in cause-effect graph method include *AND*, *OR*, *NEGATION* etc., which represent logical relationships. There are several kinds of constrains that may be applied for input conditions in the applications. For example, the exclusive operator means that in an application, two causes cannot simultaneously present. The constrains can be deduced by studying the specification.

In our unit testing model, we use similar idea to the cause-effect graph in the sense of focusing on the cause and effect relationship. The components in a class are analyzed as well as their relationships. On the other hand, it is not a simple cause-effect graph method. The unit testing method provides a set of predefined terms for the components in a unit. The following is the detailed discussion about the comparison of cause-effect graph and the unit testing model.

First, cause-effect graph is a more general testing method than unit testing modeling method. It can be used in various testing and requirement analysis applications. In contrast, the unit testing model is targeted only for the class testing in object-oriented pro-

gramming but not for any other levels of testing(at least at current stage). For the cause-effect graph method, because of the nature of the application domain, there is usually no need to analyze the cause in detail. The unit testing method focuses more on the relation among the causes like AND and OR relationship. In unit testing modeling, we need exam the causes more closely. In our model, we group the member functions into different categories. In cause-effect graph, there is no concept of *order* which means that the order of the causes is not an issue. In unit testing model, test order is introduced. The order must be used as a guideline to generate efficient test cases. Other different features include the way to analyze the attributes in a class and the object-oriented relationships such as inheritance relationship. Moreover, the unit testing model deals with the reuse issue. How to reuse unit testing model is discussed. Reuse is not a topic in cause-effect graph.

5.2 Data member and member function classification

There are various types of relations that can exist in a class. In essence, a class consists of two kinds of basic constructs(in some case, there is only one), say data members and member functions. Generally speaking, the combination of a data members represents the state of an object. Member functions provide a means for internal and external components to change the state of an object. From the structural perspective, some constrains may apply in different implementation. For example, in C++, the access control is implemented with keywords *public*, *friend*, and *private*. The structure of a class can be represented using the following graph.

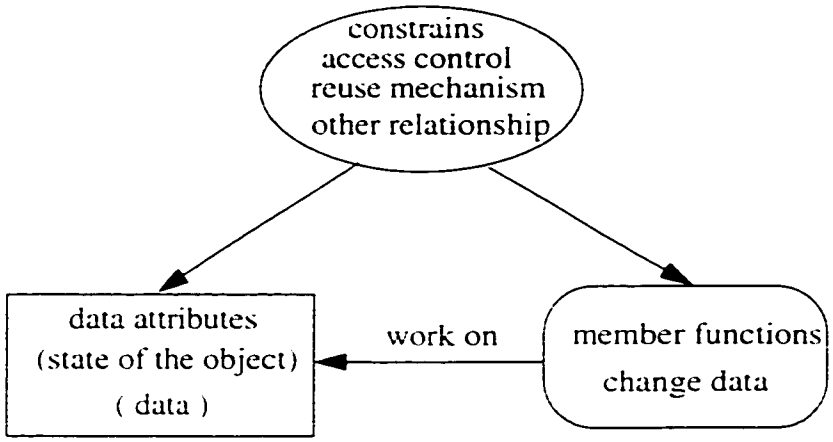

Figure 6 The high level unit representation for a class

The roles of the data in an object may be different. The data should be treated differently in testing. The data representing the state of the object are the key items in unit testing modeling. According to the general purpose of data member, we can classify the data into two groups: the *State-Related data* or *SR data* and *None-State-Related data* or *NSR data*. They are defined as follows:

SR data: *the data in this group represent the state of an object. Changing of the value of the members in this group may cause the state transition of the object.*

NSR data: *the data in this group do not represent the state of an object. These data may be used internally by the object or just to provide some information as operational measurement or other use.*

A class may not necessarily have both SR data and NSR data. Sometimes, some data are coupled together, in which case, all of them should be put into the SR data group. Also in some cases, not every change of data causes the state transition of the object. The following is a class definition:

```
Class Counter
{
    private:
       int    counter, counter1 = 0;
       int    flag = 0;
    public:
         void increase()  { counter++; counter1++;}
         void check()  { if  (counter > 10)
                          flag = 1; }
         void print_utility()  { if  (flag > 0)
                                   count >> "overflow!/n"; }
};
```

In the above example. The variable *flag*'s value determines the output. We can check *counter* directly. It is used here just as an example to show different types of variables. The *counter1* variable does not work like *counter or flag*. It is a NSR data.

To distinguish the difference between these two kinds of data, we use rectangle to represent SR data and round corner rectangle to represent NSR data. The symbolic representations of the data members in a unit will be listed in the right side of the unit testing graphs. Furthermore, we put the SR data in the upper right part of the graph. It will be beneficial if the representation of the data members also includes the detailed information of the data member. This is useful for the SR data. For SR data, we can indicate how the data represents the different state of a unit. There is no need to provide more detailed modeling symbols for this part because it depends on the nature of the application. In certain application area, more detailed rules can be applied by the users. Usually the data type, value domain partition are recommended to be included. The symbols of the two kinds of data are listed in figure 7.

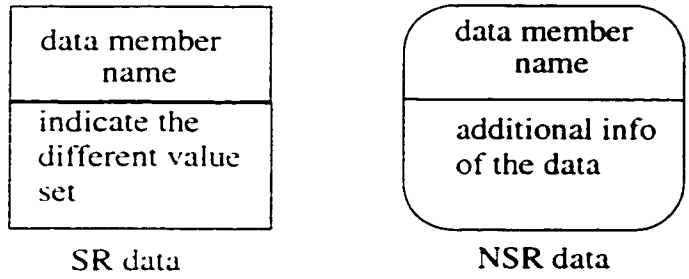

Figure 7 Representation of data members in a class

An example of the data member representation is as follows(the class *Counter* is used in the example)

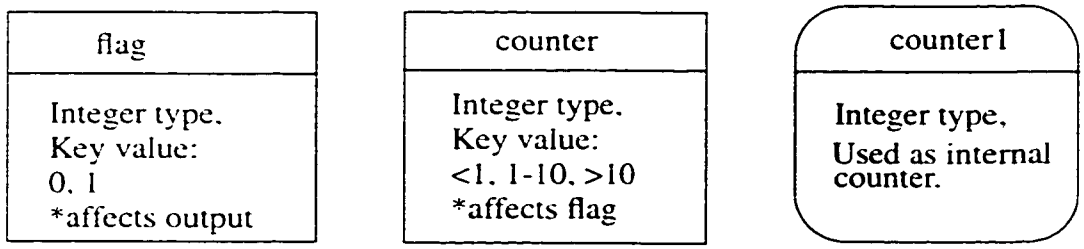

Figure 8 Data member representation of the class Counter

Correspondingly, the operations, implemented via member functions in C++, in a class can be classified based on how they process the data. Some of the member functions will process the SR data while others process NSR data, or in some cases, both. To better capture the relationship, it is necessary to examine how member functions process the data. In our unit testing model, the member functions fall into one of these groups: *Directly-State-Change* member function or *DSC* member function, *Indirectly-State-Change* member function or *ISC* member function, *Combined-State-Change* member function or *CSC* member function and *Non-State-Change* member function or *NSC* member function. In addition, the constructor and destructor in a class are labeled by *C* and *D*. The definition of these groups are as follows.

DSC: *the member functions in this group directly change the value of SR data and cause the object state change.*

ISC: *the member functions in this group indirectly change the value of SR data and cause the object state change by calling other member function(s).*

CSC: *Combined-state-change, the member functions in this group change SR data directly and via ISC functions.*

NSC: *the member functions in this group do not change the value of SR data and therefore do not cause the object state change.*

C: *Constructor of the class.*

D: *Destructor of the class.*

Like the representation of the data members in a class, we represent different member functions with the labels DSC, ISC, CSC, NSC. The member function components in a class are listed in the left side of a testing graph. For the member functions of constructor and destructor, because of the nature of the functions is different from the other kind of member functions, we have special symbols for them. The graphical representation of the member functions is shown in figure 9.

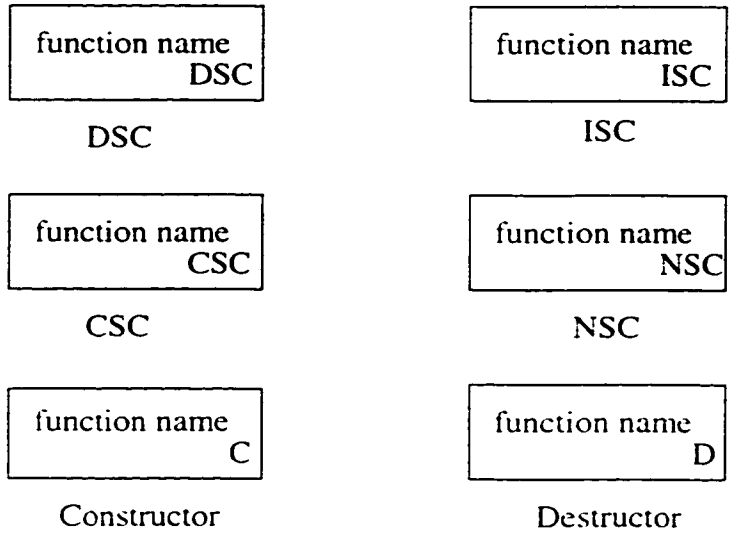

Figure 9 Representation of the member functions in a class

Taking the class definition *Counter* as an example, the member function components are listed below:

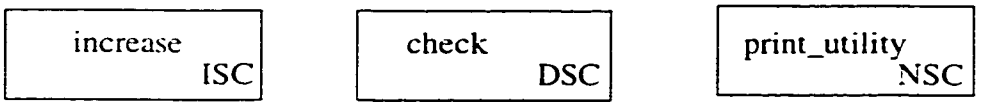

Figure 10 Member functions in class Counter

The relation between the member functions and the data members in a class is represented with connections among these components. The connections between the member functions and the data members are represented with links in our unit testing model. A link from a function to a data member means the member function has effect on the value of the data member. For the constructors in a class, although they deal with most of the data members in usual case, due to the simplicity of the functionality, the connections can be omitted for them. Therefore there is no lines from the constructors and to any data members.

For a DSC group function, there are direct links toward data members. The relationships can be one to one or one to many. Dotted links connect ISC functions with data members to show that they will change the data value of the data member. We may have a function sequence of functions. For example, function f_3 changes the value of SR variable a by calling function f_2 and in turn f_2 calls the function f_1 which changes the value directly. In this case, we draw a line from f_3 to variable a and put f_2 above it. The same thing happens to the line from f_2 to the variable a. The graphical notation and an example of the notation are shown in figure 11, 12.

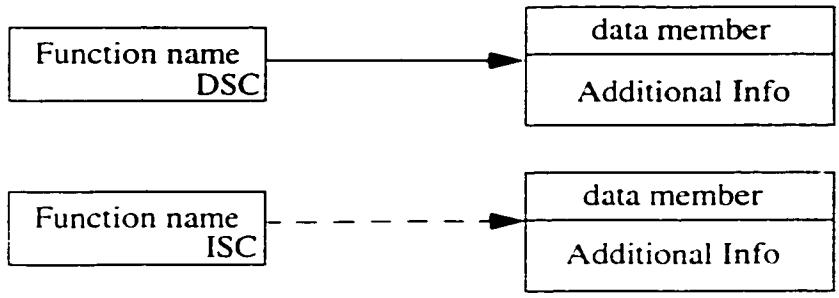

Figure 11 The connection between the member function and the data

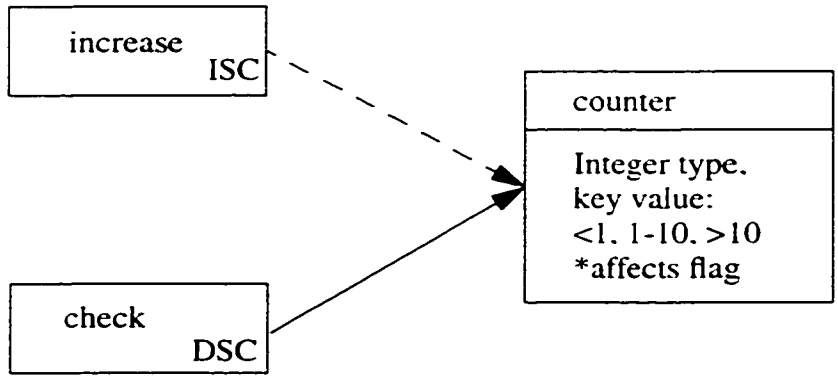

Figure 12 Example of the relations of member functions and data members

The unit testing model notations represent the components in a class and relationships between the components. By using the notations, the static structure of a class can be represented by linking the data and operation components.

The following class can be used to show that unit testing model is useful to capture the relationships in a class. The class example is from[11]. In the example, a coin box class of a vending machine is implemented in C++. It is assumed that the vending machine has very a simple functionality and the code controlling the physical device is omitted. Only

the quarters are accepted. The operations include adding a quarter, returning the current quarters, resetting and vending. There is no need to explain the detailed implementation.

```
class CoinBox
{
   unsigned totalQtrs;   // total quarters collected
   unsigned curQtrs;   // current quarters collected
   unsigned allowVend;   // 1 means vending is allowed
public:
   CoinBox()  {Reset();}
   void AddQtr();   // add a quarter
   void ReturnQtrs() {curQtrs = 0; }    return current quarters
   unsigned isAllowedVend() {return allowVend; }
   void Reset()   { totalQtrs = 0; allowVend = 0; curQtrs = 0; }
   void  Vend();     //if vending allowed, update totalQtrs and
                     //curQtrs
};

void CoinBox::AddQtr()
{
   curQtrs = curQtrs + 1;  // add a quarter
   if ( curQtrs > 1 )   // if more than one quarter is collected,
   allowVend = 1;   // then set allowVend
}

void CoinBox::Vend()
{
   if ( isallowedVend())  // if allowVend
   {
      totalQtrs = totalQtrs + curQtrs;   // update totalQtrs,
      curQtrs = 0;                       // curQtrs, and
      allowVend = 0;                     // allowVend,
   }
}
```

In the above example, there are three data members in the class definition. The data member *allowVend* is the most important one, which indicates whether the machine is allowed to vend a drink. The data member *totalQtrs* is used to keep track of the total quarters that the machine has received, and *currentQtrs* is the data member that has indirect impact on the machine states. As we discussed before, we treat it as SR data.

The member functions in this example include the constructor and other utility functions. The *isAllowedVend* function has no impact on the data members. Others like *Reset* has impact on the both SR and NSR data. The corresponding testing graph is shown in Figure 13. Usually there is no lines pointing to the data members from constructor of the class. In this example, in order to show the indirect change of the data member, dot lines are added for the constructor.

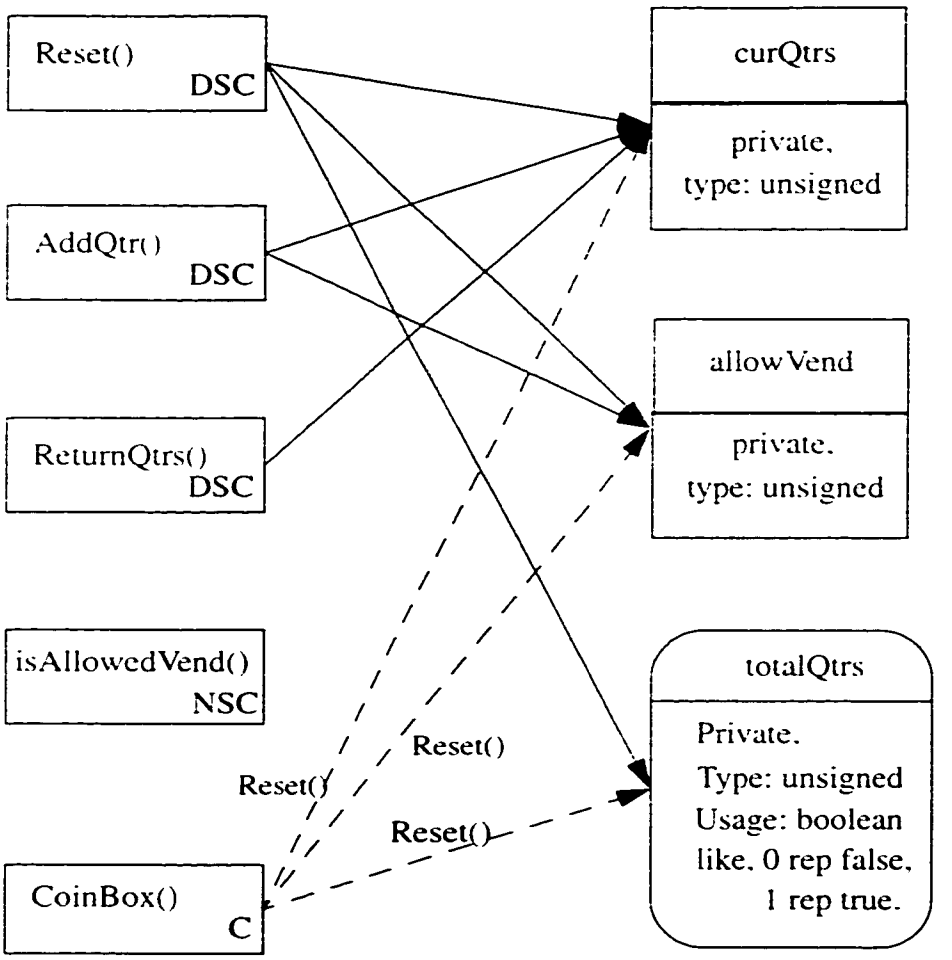

Figure 13 Testing graph for the example class CoinBox

5.3 Relationship discussion

5.3.1 AND and OR relationships in a class

There are different types of relationship in a class. We develop a concept called *AND* relationship in unit testing modeling. Unlike the conventional logic relationship, we do not expect the *NOT*, *NOR* or *NAND* relations in a class. In general, the AND operator means

that when all the input is TRUE, the output is TRUE, otherwise, the output is FALSE. It is not necessary to make a mapping from the conventional logic AND concept to the relation in a class. Usually the AND relation can be applied in *member function chains*. Correspondingly, in our discussion on class structure, the AND relation is defined as

> *If there is a member function chain C for a data member in a class, then all the member functions in the chain have the AND relationship for that data member. A member function chain is a series of member function calls to finally change a data member. The relation can be represented as*
>
> **A * B *... * C * D -> X**
>
> *Where A, B, C, D are member functions in the class and X is a data member. A is the starting point of the chain.*

The AND relation introduced here is not the same as that in logic. The commutative property is not supported, i.e., we have

A * B is not equivalent to B * A

The above discussion presents a formal method to capture the member function chain in object-oriented system. This is useful when we think about the causes of a data member change.

On the other hand, OR relationship among the member functions has a very similar behavior like the OR operator in general logic. In our unit testing model, The OR relation can be defined as:

> *Assume a group of member functions G in a class for a data member in which each member function can change the data member independently. The member functions in G have the OR relationship. The relation can be represented as*
>
> **A + B + ... + C + D -> X**

Where A, B, C, D are member functions in the class and X is the data member that can be changed by the member functions.

The OR relation is the same as that in logic. Commutative property is supported.

A + B is equal to B + A

OR means if one of the member functions is called somewhere, the value of the data member may be changed. By grouping the functions that work on the same data member together, we can find the functions that have impact on the object state easier.

By our definition AND and OR are basic relations in class structure. This is a relatively formal way to express the relations in a class. It provides information for testing. Part of the results to apply this definition to the ***CoinBox*** class example is shown as follows:

*CoinBox * Reset -> totalQtrs, curQtrs, allowVend*

CoinBox + Reset + AddQtr + ReturnQtrs + Vend -> curQtrs

addQtrs + Vend -> allowVend

5.3.2 Test order

A property of a member function with respect to a data member is the *level* of the member function. The *level* indicates the order of the member functions accessing the data members. For example, if function f_2 changes the value of data member *A* via function f_1 then we consider the level of f_2 is 2. Level analysis implies that before f_2 is tested, f_1 must be tested. The concept of level gives us the useful way to capture the test order. The notion of the test order is defined as follows:

Given a data member D in a class,

- *if member function F changes D directly, the test order of F is 1.*

- *If member function1 indirectly changes D via member function2 then the test order of member function1 equals to the test order of member function2 plus 1, that is,*

$$T1 = T2 + 1$$

*where **T1** is the test order for function1 and **T2** is the test order for function2. Please note that **T2** may not be a single value. It may be a set of values. In this case, the '+' operator applies to all the elements in the set.*

- *Those NSC functions have test order 0.*

The test order is related to the data member and member functions. It means that a member function may have different test orders for different data members. Moreover, even for the same data member, the test order may be different for different member function chains. For example, we may have the following relationships,

1. A * B * C -> M
2. A * E -> M

the test order of A for M is the set S = { 2, 3 }. In the first relationship A has a test order 3 and for the second relationship, A has a test order 2. These test orders should be covered in unit testing.

Assume a class and the following relationships,

1. A * B * C * D -> X, Y, Z
2. A * E -> X, Z
3. A + B + F -> Z

Considering functions A, B, C, D, E and data members X, Y, Z, for the data member X, the first relationship implies that A has test order 4, the second relationship implies A has test order 2. For the data member Z, we have to be careful because the relationship 3 is OR relation. Test order cannot be derived from an OR relationship only. The reason is that an OR relationship only shows which member functions have impact on the data members. But it does not show how a member function changes the data member. Therefore there is no test order information.

By the above discussion, the test order of a member function should be a range. For a single value, we take it as a one element set. The elements in the range are not necessarily consecutive. In general, we have the following result.

> *Let S be the set of member functions. We define the operator '+' applies to all the elements in the set S. In a member function OR relationship ... F_i + F_i ... -> D. S_i may not equal to S_i + 1.*

The concept of test order is useful when test cases are generated for a class. It provides additional information to guide test case generation. Without a systematic discussion of the test order, it is not easy to better cover the test cases.

5.4 Test case generation

We have presented some basic aspects of unit testing modeling method to capture the structure of a class. The unit testing model is used to generate test cases. By the above discussion, we can see that the unit testing model contains a set of graphical notations and symbolic definitions to represent a class in an object-oriented system. We now discuss the guidelines for generating test cases using the model. We assume that a class testing graph has been set up according to the analysis of the class structure and specifications.

Rule 1: Given a class, the constructor(s) (C function) should be tested first.

Explanation: A constructor is a member function that initiates an object of a class. To use an object created from a class, a user has to initialize the object. The test cases make sure that the objects are created correctly. Usually, there are more than one constructor in a class. The test order of the constructors is 1, but there are exceptions, like the class *CoinBox* example.

Rule 2: Given a class, ***the destructor (D function) should be tested last.***

Explanation: The destructor(if there is one) should be exercised to confirm that the objects are cleaned after their use. Some languages such as Smalltalk do not have destructors and they depend on the system utility for garbage collection. After tested all the aspects of the behaviors, we need to test wether an object is destroyed correctly after use.

Rule 3: For a given class, the test should be conducted in different levels from the lowest test order to the highest test order.

Explanation: For a given class, to organize the test, we calculate the test orders for member functions. Once we have calculated all the test orders for all the member functions, we can conduct the test from the lowest level to the highest test order. Thus a designer/tester can test the software more efficiently. We will not have a test case that fails because of the failure of lower level function calls. It provides the designer/tester with an efficient and manageable way to deal with the member function chain testing. The test order calculation applies to the DSC, ISC, CSC, NSC member functions in a class. The AND and OR relationships provide additional information when the test order is calculated. When the test order of a member function has more than one elements, for example, S = { s_1, s_2... s_k}, it means the member function is in more than one member function chains. The number of the member function chains is greater or equal to the number of the elements in the test order set S. For each element in the range, the member function should be tested because this member function may belong to different member function chain and the function behaves differently when it is in different chain. For example, we have A * B * C * D -> M and A * F -> M. In this case S = { 2, 4 }. We have to test both scenarios for full coverage. This is similar to the path coverage or statement cov-

erage in traditional testing method. We need to cover all the member function chains for a member function.

Rule 4: ***For a given class, the DSC, ISC, CSC member functions should be fully covered, which means that all the state transitions that one object can have should be exercised.***

Explanation: The DSC, ISC, CSC member functions are the functions that can change the state of an object. All these member functions should be covered. Furthermore, all the allowed state transitions should be exercised at least once. In our discussion, we do not focus on the model aspect that captures the state transition of a class. For now, we can use the general computer science concept of state machine.

In the above discussion, we do not mention some of the object-oriented features such as inheritance and polymorphism in a class. The class we discussed is about base class or component structure of a superclass. The classes with other features will be discussed later in the thesis. The following is the algorithm for generating test cases for a class.

Algorithm 1: Generate test cases for a (base) class

Given a class A,

Begin

Group the data members in A according to the design and specification,

Identify the member functions using the definitions(DSC, ISC,..),

Put the corresponding member functions near the data members if it is possible,

Draw the test graph,

Capture the AND and OR relationships of the member functions,

Compute the test order.

Generate test cases based on the developed object-oriented unit testing rules.

End

5.5 Discussion about attribute in more complex class

The above discussion includes classification of data attributes, member functions and the process of test case generation. We focused on the classes that have a simple structure. We assume class state that represented by the SR data attributes consist of a group irrelevant attributes. In a real application, the structure of a class might be more complex. For example, a state of an object may be represented by a set of data members but not a single attribute. An example class is as follows:

```
Class Example
{
      int a, b, c;
   public:
      void Example () { a = b = c = 0; }  // constructor
      void ^Example() {}    //destructor
      void increment(int & i) { i++; }  //  incrementing
      void decrement(int & j) { j--; }  // decrementing
      int f() { printf("the result now is: %d.\n", c = a*b); };
      int g() { if  ( a * b > 0 )
                  cout << "now the data are in the same signal.\n";
                elseif ( a * b == 0 )
                  cout <<"at least one of the data is zero.\n";
                else
                  cout <<"one of the data is negative.\n";
              };
};
```

In the class definition, the object of the class provides different outputs depending on the status of a set of data attributes *a* and *b*.

In a more general description, a class has a set of attributes { a, a_1, a_2,.... a_x}. The subset { a_i,..... a_n} can have impact on the state of the class. The change of a single attribute a_k does not represent the state change of the object. There are member functions related to the subset of the attributes. The functions that have impact on the subset can be classified by the developed method. We should treat this type of subset of attributes as "one attribute" from the testing point of view. To make the classification easier, a new type of pseudo-attribute has been defined as follows:

PSR data: *the data in this group represents the state of an object. The data is not a real attribute in a class. It consists of more than one data attributes. The change of the members in this group causes the state transition of the object.*

It is clear that the behavior of PSR data is similar to SR data. In a test graph, we may use PSR data. To distinguish the PSR data and SR data, we adopt the symbol(dotted frame) for PSR data as follows:

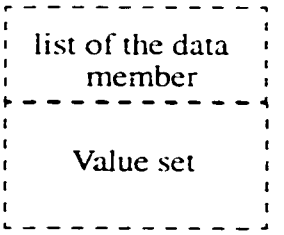

Figure 14 Graph representation of PSR data

5.6 Discussion on member functions

Relationships of member functions and the data attributes sometimes can be one-to-many instead of just one-to-one. The previously defined rules can be used as guidelines for the general class definition. Here, we introduce the graphical notation for the one to many relationship of the member functions and the data attributes.

A member function can take more than one attribute as parameters. Therefore, the mapping could be one to many in a test graph, and the test order may be different too. An abstraction of the situation is shown in figure 15.

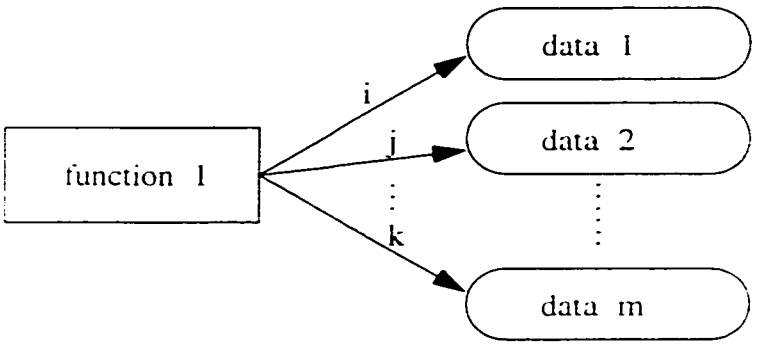

Figure 15 Member function has more than one level data attributes

In the above test graph, $i, j, \ldots, k$ are test orders related to the data members. We assume that test order $i < j < \ldots < k$. From testing perspective, there is no need to pay special attention to the one-to-many relationship. We introduce the representation here to complete the set of notations.

5.7 Inheritance in modeling (graph reuse)

Inheritance is one of the most important object oriented features that can reduce the design work remarkably. How to test this mostly used feature is an important issue. An advantage of testing inheritance relationship is that we can reduce the time to develop test cases for the subclasses.

In C++, when a subclass is created, it inherits all the attributes and operations from the superclass. As a new class, the subclass can define its own attributes and operations. If the superclass has virtual member function, the subclass has the implementation of the virtual function and it may differ from the implementation of the superclass. The newly introduced features work on the new or inherited attributes. According to [21], software reuse

does not resolve the problem of testing the subclass. It may make testing object-oriented software more complex. We have to retest the subclass anyway. In this section we discuss how to maximize the reuse of the test from superclass.

Test order is a part of the guideline for a tester to perform testing. If new operations are introduced in subclass, they have their own test orders and the test orders should be considered in the new subclass testing. To discuss the relationship of the operations and the data members in a subclass, we need to investigate the inheritance in subclass testing. From the object-oriented design perspective, inheritance gives the designers ability to reuse the design of the superclass. Both attributes and member functions in superclass are reusable. There are different implementations for inheritance relationship in object-oriented design. For example, in C++, the declaration of a subclass can have three different types of access controls, which are part of inheritance mechanism, namely *private*, *protected*, and *public*. The protected segment in a class gives the flexibility of accessing that part from a subclass to increment the degree of reuse. In contrast, in Smalltalk, there is no such kind of mechanism. Another example could be the inheritance hierarchy. C++ supports multiple inheritance, which means a subclass can inherit from more than one superclass. However, Smalltalk supports single inheritance only. While multiple inheritance gives the flexibility of reuse, it also makes testing very complex.

Since inheritance has various implementations, a high level discussion will cover most languages. C++'s approach to inheritance will be selected if a detailed discussion of implementation is required.

Given the mechanism of inheritance, different designs of a subclass can be proposed. Reuse depends on the design of a class. It is not always the case that a designer can reuse a lot from a superclass. For example, all the member functions in a subclass can be newly defined or implemented differently. In this case, from the testing point of view, there is no difference between this subclass and a newly defined class. The designer has to test it from the scratch. In another case, if there is little interaction between the newly introduced member function and the old design, the designer will take advantage of the test reuse. To make the discussion more accurate, the classification of the newly added operations are listed below. To simplify the discussion, we assume that the attributes in the superclass

have the same status in the subclass, which means that SR attributes in the superclass are also SR attributes in subclass. NSR attributes in the superclass are also NSR attributes in the subclass. In the applications, this is usually the case.

Reuse case1: Newly defined attributes

Reuse: the subclass may introduce new attributes. The newly defined attributes will not come by themselves. The subclass may define or redefine member functions for the attributes. In either case, the new attributes and the member functions can be simply added to the test graph of the superclass using the previous developed method. The redefined member functions should override the old one in the test graph.

Reuse case 2: No new attributes

Reuse: Subclass may define its own member functions for the application. The subclass may have new member functions operating on the attributes inherited form the superclass. It may override member functions to implement different functionality. In both cases, the test graph can be reused. For the new member function, nodes are added to the existing graph using the developed method. For the overriding member functions, only those functions which depend on the overriding functions need to be changed. Virtual functions can be added with this method.

Reuse case3: Overridden attributes

Reuse: The subclass may override the attributes in a superclass. As discussed above, there may be new member functions or overridden member functions for them. The defined or redefined member functions can be added to the graph to generate test cases using the previously developed method.

Virtual functions are the functions that may have different implementations in subclasses. The superclass is called *abstract class* if there is no implementation for a virtual function.

Abstract superclass can not be instantiated because it has no implementation for some of its member functions. For the virtual functions, we mark them using the key word *virtual*. There is no connection between a virtual function and data members until there is a real implementation for it.

From the above discussion about inheritance, we can see that reuse can be divided into two cases, one is a subclass that introduces new data members and has newly defined member functions for them. In this case, the new test graph is obtained by combining the added data members and member functions with the existing test graph. In the other case, the test graph will be modified using the existing graph.

Algorithm 2: Generate test cases for a subclass:

Given a subclass B inherited from class A
for all the attributes, do:

If ***the attribute and the member function that operates on it are not changed in the subclass, keep the corresponding part of graph in the new graph.***

elseif ***the member function which works on the attribute has been changed, replace this member function node with the new node.(this is the case of overriding)***

elseif ***the attribute is overridden attribute, use the new attribute's relation.***

elseif ***the class B has newly defined attributes and member functions which operate only on these attributes, generate the test graph using algorithm 1. Combine the new graph with the test graph for class A to get the graph for class B.***

enddo
end;

5.8 Discussion about multiple inheritance

Multiple inheritance means the ability of reusing more than one class. It gives the designer flexibility in the design. However, multiple inheritance also introduces issues in testing. For example, how to test the multiple inheritance using the developed method and how to reuse the test graph in a subclass, which inherits from more than one class.

The multiple inheritance in OMT is shown in the following diagram:

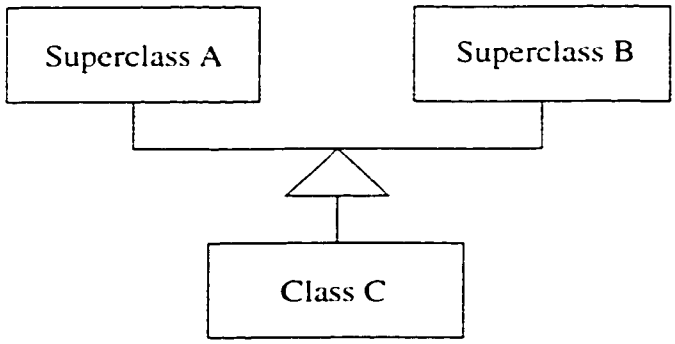

Figure 16 Multiple inheritance representation in OMT

We now discuss the general idea in subclass testing with multiple inheritance relationship.

Multiple inheritance testing

Assuming class C inherits from more than one class,

Begin

Test all non-abstract superclass(es) using algorithm 1 or 2

Generate test case using algorithm 2 for C

For each of the inherited member functions, if a member function has different implementation in different superclasses, add test cases to make sure that the desired function is called from one of the superclass.

End

We can apply algorithm 2 to multiple inheritance relationship in most case. In addition, for multiple inheritance, we have to think about the case that the superclasses have similar functions. For example, assume class *PC* and *MAC* have member function *print* with different implementation. Assume class *POWERPC* is a subclass of *PC* and *MAC*. When the *print* function in *POWERPC* is called, for *PC* format graph, we need the function from

class *PC* and for *MAC* format graph, function from *MAC* class is required. We need test cases to cover this scenario.

5.9 Discussion on polymorphism

Polymorphism is another key concept in object-oriented design. It means a component like member function or operator has the ability to take more than one form. There are different kinds of polymorphism such as function overloading and parameterized classes. We constrain our discussion on overloading type polymorphism. Polymorphism provides a language with a nice interface that makes the software more consistent and easy to understand from the application perspective. A common example of polymorphism is shown as follows:

```
Class Shape
{
        //     Some definition here,
        virtual print() = 0; // prototype for later use
};

Class triangle : public class shape
{
        //     Some definition here,
        print() {  // implementation of printing a triangle here
                                ...
                };
};

Class circle : public class shape
{
         //    Some definition here,
         print() {  // implementation of printing a circle here
                                ...
                };
};
```

The implementation of function *print* is different in different subclasses. In an application, which implementation is used depends on which object calls the function. In our example, if an object of class *Shape* is a triangle, then the *print* function call uses the implementation defined in subclass *triangle*.

Inheritance and overloading may have different impact on polymorphism. For example, in the previous example, the *print* functions of two subclasses are all inherited from the same superclass. They are horizontal. On the other hand, overridden function in an inheritance hierarchy is the example of vertical relation. If a testing method does not capture this kind of structure, it is error prone to generate test case and some of the possible tests may be missing. As complimentary part of the unit testing modeling, a table is defined to capture the polymorphism relationship in a class. Here we constrain our discussion to polymorphism on the virtual functions. Other types of polymorphism can be expanded and examined in future research.

The polymorphism feature of member function is captured using a polymorphism table. In this table, each of the member functions is labelled with the class name that it is associated with. Although the polymorphism relationships look like that of class inheritance structure, they are different. This is one of the reasons that we create a new table instead of using a class inheritance model from other modeling tools. The title line of the table is the function name and the prototype. Under the title line is the relation structure. An example is shown in figure 17 and table 1. The class *Object* is the superclass and has *Graph* and *Text* as subclasses. Both *Graph* and *Text* have their own subclasses as shown in Figure 17. We assume all the classes have member function *print*. The symbol "(v)" means the member function is a virtual function and "*" means no implementation(reuse from the superclass). From table 1, we can immediately see that there are four implementations of print function to be tested. They are from classes *TRIANGLE*, *ELLIPSE*, *COLOR_CIRCULE*, *ITALIC*. As contrast, there is not much information for testing member function *print* by a glance at figure 17.

More complex polymorphism needs more investigation. Polymorphism is difficult to test. We provide a means to visualize the relation to provide the tester with more information. The rule of thumb for polymorphism testing contains two rules:

> ***Rule 1,*** *All the implementations for a member function should be tested*
>
> ***Rule 2,*** *All the scenarios should be covered to make sure the right implementation is invoked in the scenarios.*

In the example, the test of the member function print include test all the implementations in classes *TRIANGLE, ELLIPSE, COLOR_CIRCULE, ITALIC*(rule 1) and for different objects, test the right member function is invoked.

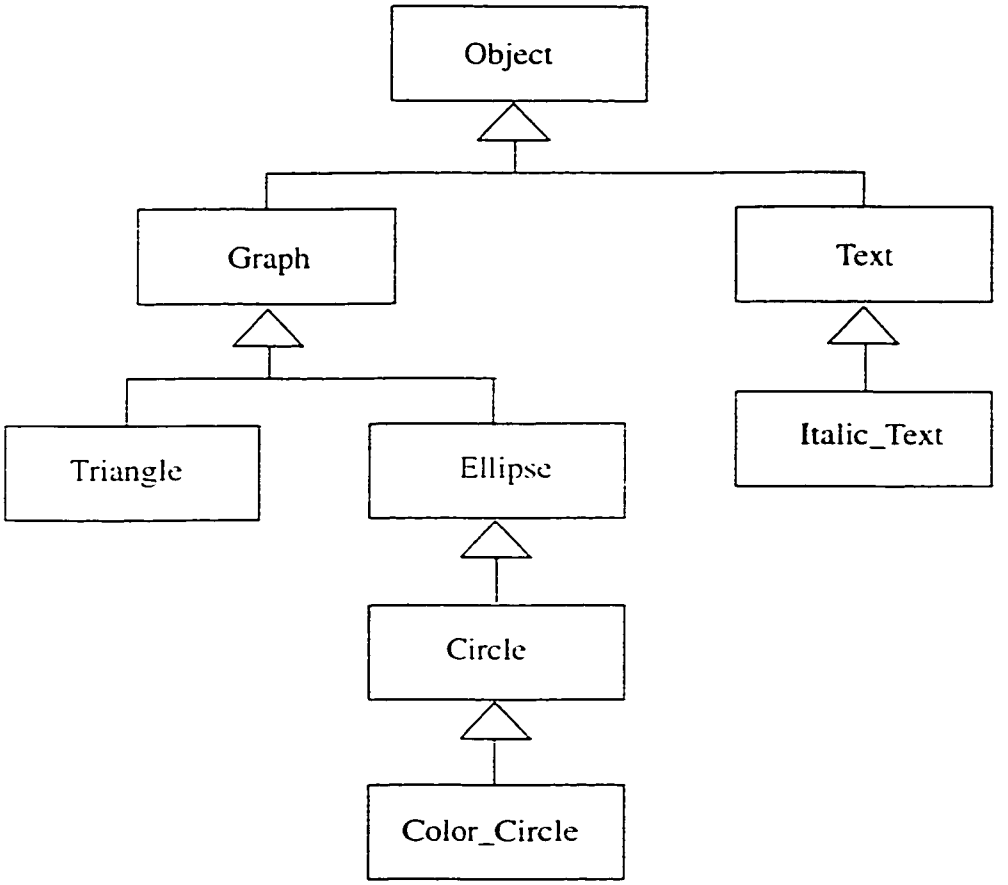

Figure 17 Inheritance graph

TABLE 1. Polymorphism of example print function

print(OBJECT)		
GRAPH(v)		TEXT(v)
TRIANGLE	ELLIPSE	ITALIC
	CIRCULE*	
	COLOR_CIRCULE	

5.10 Advantage and limitation

Class is at the lowest level in an object-oriented design. From a high level view, classes have similar structures. But the functionality of classes may differ from each other for different applications or different parts of the same application. Therefore, classes have different behaviors and properties according to the design. For example, some components in

an application are used as control elements while others may be used as storages. From the testing point of view, it is difficult to develop a general unit testing model that best fits for all types of classes. This section discusses about how our unit testing method can be used for different types of class.

Base class also depends on its subclass because it may have virtual functions which needs the implementation from the subclass.

5.10.1 Abstract class

Abstract class is a type of superclass in object-oriented system. It is normally used as an interface and has no implementation for some member functions. In C++, abstract class is a class with pure virtual function. All of the member function in a class may be pure virtual functions. An abstract class can be customized in several ways. An abstract class cannot be used in an application without instantiation. In a test graph, if the class is an abstract class, the graph representation of virtual functions should include information indicating that the function is a virtual function. Furthermore, if the implementation detail is unknown, the function classification (ISC in the example) can be omitted.

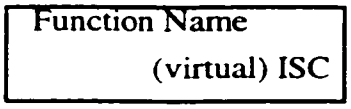

Figure 18 Virtual function representation in unit testing model

Without the implementation, we do not know how to test the virtual function. Therefore, a virtual function in a testing graph will not be taken into consideration in test case generation. The test order of the virtual functions can be omitted as well. If a superclass has only a few virtual functions, testing can still be conducted for the class to achieve high coverage for the existing functionality. If most of the member functions are pure virtual functions, although a test graph may be created for the superclass, the graph is not used for generating test case. It is used for providing information for the subclass test graph generation. If most of the member functions in the superclass are to be pure virtual functions and subclasses have different application intentions, the subclass test graphs may differ

from each other. Note that even a member function is not a virtual function, it is also possible that the function is overridden in the subclasses. How to handle virtual functions and overridden functions have been discussed in section 5.9. In abstract superclass, there is usually no constructor. Also, if most of the member functions are virtual functions, data members may not be presented in the class definition. In this case, the class appears as an interface.

5.10.2 Node class

A node class is a class in the class hierarchy between the superclass and the implementation subclass. A node class is usually viewed as an extension of the abstract superclass which is based on the service provided by the superclass. A node class also provides some extended services that are not provided by the superclass. It provides some services that can be refined by subclasses. An example of node class is shown in figure 19. The node class can be instantiated. From the property of the node class, we can see that the test graph should be generated for the node class.

A node class may have nontrivial constructor while its abstract superclass usually does not have. The unit test modeling method fits for node class very well. During the unit testing phase, test graph as well as the real test cases should be generated for the node class.

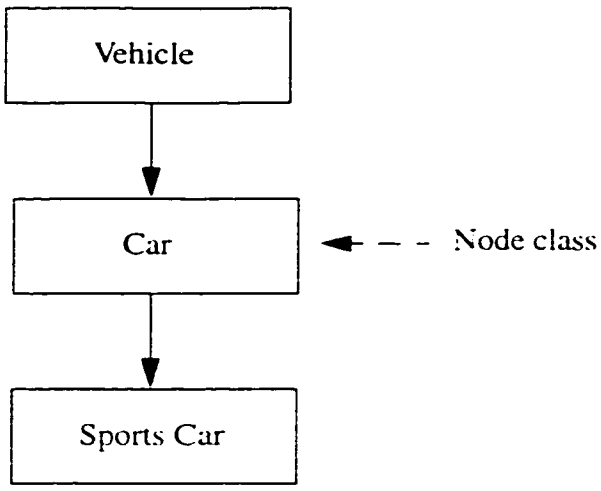

Figure 19 Example of node class

5.10.3 Interface class

An abstract superclass represents an abstract type in the system, which is a kind of interface class. There are other types of interface class. In this section, we discuss the general interface class, which has no specific function and behaves. Although the interface class provides only an interface, it is important to understand the testing issue for this kind of class.

Sometimes, a subclass needs an interface class to refine or implement the functionality. One of such cases is the name clashing in multiple inheritance example. The following is an example of interface class.

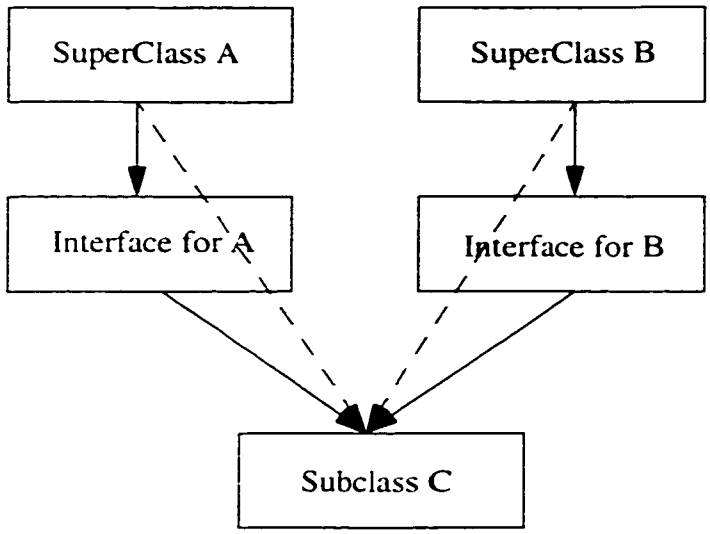

Figure 20 Example of the usage of interface class

In the example, we assume superclass A and superclass B have virtual functions that have the same name but different functionality. Also it is not appropriate to refine the two virtual functions within one function because the two functions in superclasses are irrelevant. Interface classes are needed here for both superclass A and B. In interface class A and B, the functions that have the same name in superclasses can be delegated by different function name and then inherited and refined by the subclass C. If the virtual functions have different parameter types, the problem may be resolved by the usual overloading rules. In general, an interface class provides a mechanism to enhance the design and provides a work around for the problems like name clashing.

An example in C++ is shown below. By introducing a new name *new_print*, we can get rid of the name conflicting problem in multiple inheritance.

```
Class Shape
{
        //     Some definition here,
        virtual void print();
};
```

```
Class S_interface : public class shape
{
        virtual new_print() = 0;
        void print() { new_print() };
};

Class Subclass : public class S_interface, ...
{
          //    Some definition here,
         new_print();
         ...
};
```

The developed unit testing method can be applied to interface class. All the rules and representations can be used for this type of class. However, the usage of interface class needs to be investigated in more detail when testing is conducted. If the purpose of an interface class is to do a transformation and provide clear interface, it should use virtual type functions. In this case, there is no need to consider the test graph for it and the test can be postponed to the subclass. The relation among of the implementation class, interface class and the top superclass should be remembered. The formal symbol for capturing the relation may be developed in the future research.

5.10.4 Handle class

While abstract class is a kind of interface class, the relation of the abstract class to its implementation is fixed. Interface class discussed in the above section is intended for refining the structure to provide a clearer interface for inheritance. There is also another kind of interface class, which is called handle class. A handle class provides a general interface for a family of classes, which may be derived from the same base class. Users of the handle class need not to have any knowledge about the representation of the class. They can use the handle class directly. A mechanism is implemented by using a pointer between the handle class and the representation class. The relationship of the handle class and the representation class is shown in Figure 20. The handle interface and representation

classes are instantiated. The handle objects contain the pointers to the representation classes. The other components in the system can access the representation using the handle objects.

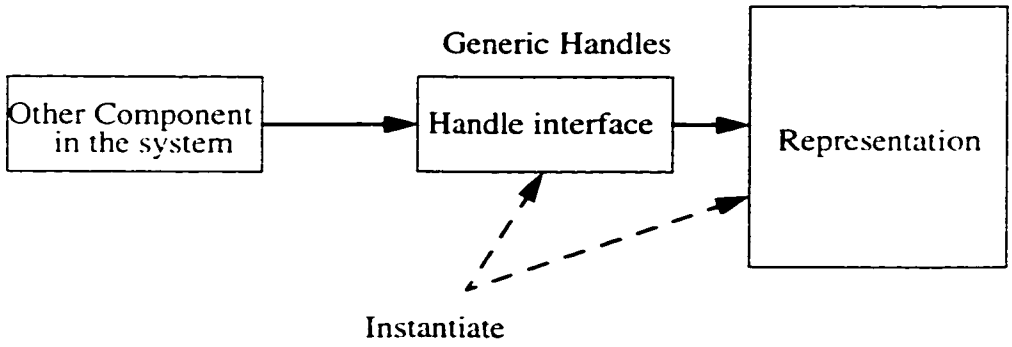

Figure 21 Handle class representation

Handle class should be small but generic for a group of classes. The representation class has the general class structure which holds the state. Usually there is no need to reuse the handle class.

Because of the nature of handle class, it is not necessary to test handle class. The class structure is relatively simple, and the test graph may not be needed. It should be considered when the representation class is tested. From the handle class perspective, the test consideration includes the constructor and overloaded operators, which implement the connection between the handle and the representation.

Chapter6

Conclusions and Future Research

6.1 Future research

As can be seen in the previous chapters, a new unit testing model has been proposed. The discussion includes the basic concepts of the unit testing model for object-oriented software, the method to capture various relationships in a class and the test case generation with the unit testing model. Future research and development will be focused on the enhancement and addition to the model. These include the following aspects.

For now the components in the unit testing model are designed for capturing the static structure of units in object-oriented system. Fully representing an object-oriented system is not easy. Other kinds of models may be needed in test case generation. For example, the current unit testing model borrows the state machine concepts. A model that captures the state transition can be developed as a compensation of the unit testing model. The state model should be similar to that of Rambough's dynamic model, which is heavily based on the state machine concept as well.

More detailed representation and effective testing method for polymorphism is one of the future research topic. Polymorphism is flexible in the design but introduces complex relationship. There is

no easy way to express it at high level. A suggested approach to handle polymorphism is to consolidate the relationship, that is, analyzing the possible polymorphism relationships, putting them into different categories, and then discussing the testing methods for each category.

As a modeling method, the developed unit testing model is based on the general discussion about class, the unit in an object-oriented system. The model should be kept as a live model instead of a fixed model. The model should be expanded when there is need to capture new relationship. Also, for the relationship we discussed, customizing is recommended when appropriate. For example, the member function representation can contain different information for different applications.

6.2 Conclusions

The thesis works on the testing issue in object-oriented software development. Testing plays an important role in software development life cycle. Testing is never an easy work. Object-oriented approach introduces more issues in software testing.

The thesis focuses on the unit testing method, i.e. class testing method in object-oriented approach. A unit testing model is suggested for unit testing. The current popular approaches in object-oriented design include different modeling methods to represent an object-oriented system. The models are used to provide the designer with a relatively systematic method to develop software in object-oriented paradigm. For applications, the models are proved useful. There is a trend to combine them together to form a more complete set. The idea to use object modeling as the engine for object-oriented development has achieved such a success that it is natural to extend the idea to the other aspect of object-oriented approach such as testing. From the language perspective, the object-oriented programming paradigm requires more discipline and work than the traditional structured programming paradigm. The thesis work proposed a unit testing modeling for object-oriented testing. Many aspects of the method have been discussed. New concepts are introduced in the object-oriented logical relation discussion. The discussion about the

new modeling method covers most of the important aspects of the testing in object-oriented development, which are listed in the following:

- *The unit representation, including the unit component representation and relationship representation*
- *Testing order*
- *Logical relationship discussion and rules*
- *Object-oriented relationship including inheritance and polymorphism etc.*
- *Algorithms*
- *Applicability discussion*

The object-oriented testing follows development. Object-oriented approach itself is far from mature. There is no systematic theory about the approach. Therefore, for the current object-oriented research, a testing model is important. The benefits of having such a model on a high level include:

- *to represent the unit in object-oriented system in a formal way*
- *the graphical notation is easy to understand*
- *easy and fast to use*
- *extendable*
- *make the object modeling a more complete set*
- *structure fits the object-oriented structure*

To conclude, the thesis proposes a new modeling method for unit testing in object-oriented development. The existing and new issues are discussed. It provides an formal and easy way to address the testing issue and can be used as complementary part in the overall object-oriented modeling method.

REFERENCES

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, "Validation, verification and testing of computer software", ACM Computing Survey, vol. 14, no.2, June, pp.159-192, 1982.

[2] B. W. Boehm, "A spiral model of software development and enhancement," ACM SIGSOFT Software Engineering Notes, vol.11, pp.14-24, Aug., 1986.

[3] T. J. Cheatham and J. Mellinger, "Test Object-Oriented software systems," In Proc. 8thAnnual Computer Science Conference, ACM, pp. 161-165, Feb., 1990.

[4] S. P. Fieldler, "Object-Oriented Unit Testing," HP Journal, vol. 40, pp.69-74, 1989.

[5] P. G. Frankel and E. J. Weyuker, "A formal analysis of the fault-detecting ability of testing methods, " IEEE Trans. Software Eng., vol. 19, no. 5, pp. 202-213, Mar., 1993.

[6] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence,"IEEE Trans., Software Eng., vol. 16, np.12, pp. 1402-1411, Dec., 1990.

[7] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick, "Incremental testing of object - oriented class structures," In Proc., 14th International Conference on Software Engineering, 1992.

[8] P. Henderson, Specifications and Programs, Software-Requirements, Specification and Testing, Blackwell Scientific Publications.

[9] M. A. Hennell, D. Hedley, and I. J. Riddell, "Assessing a class of software tools," In Proc. 7th Intern. Conference on Software Engineering, IEEE, Mar., pp. 266-277, 1984.

[10] D. Hoffman and P. Strooper, "The testgraph methodology: automated testing of collection classes," J. of Object Oriented Programming, pp. 35-41, Nov.-Dec. 1995.

[11] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y. Kim, Y. Song "Developing an Object Oriented Software Testing and Maintenance Environment," Commu. of the ACM, vol., 38 no. 10 Oct., 1995.

[12] T. J. McCabe, "A complexity measure," IEEE Trans. Software Eng., vol. SE-2, no. 4, pp. 308-320, Dec., 1976.

[13] J. D. McGregor and T. D. Korson, "Integrated object-oriented testing and development process,"Commu. of ACM, vol. 37, no. 9, pp. 59-77, Sept., 1994.

[14] B. Meyers, "Object oriented software structure," Prentice_Hall, 1992.

[15] L. J. Morell, "A theory of fault-based testing," IEEE Trans. Software Eng., vol. 16. pp. 844-857, Aug 1990.

[16] J. D. Musa, "A theory of software reliability and its application," IEEE Trans. Software Eng., vol. SE-1, no. 3, pp.312-327, Sept., 1975.

[17] J. D. Musa and K. Okumoto, "A comparison of time domains for software reliability models" J. Syst. Software, vol. 4 pp. 277-287, 1984.

[18] G. J. Myers, The Art of Software Testing, Wiley, 1979.

[19] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating functional tests," Commu. of ACM, vol. 31, no. 6 pp. 676-686, June 1988.

[20] S. M. Ross, "Software reliability: The stopping rule problem," IEEE Trans. Software Eng., vol. SE-11, no. 12, pp. 1472-1476, Dec., 1985.

[21] D. E. Perry and G. E. Kaiser, "Adequate testing and Object Oriented Programming," J. Object-Oriented Programming, pp. 13-19, 1990.

[22] R. E. Prather, "Theory of program testing - an overview," The Bell System Technical Journal, vol. 62, no. 10(ii), Dec., 1983.

[23] PSC Training Note, 1995

[24] J. A. Purchase and R. L. Winder, "Debugging tools for object oriented programs," J. of Object Oriented Programming, vol., SE-13, no. 7, pp. 761-765, July, 1987.

[25] J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, W. Loresen, "Object oriented modeling and design" Prentice Hall, 1991.

[26] N. R. Saxena, E. J. mcCluskey, "Linear complexity assertions for sorting," IEEE Trans. Software Eng., vol. 20, no. 6, pp.424-431, June 1992.

[27] N. D. Singpurwalla, "Determining an optimal time interval for testing and debugging software," IEEE Trans. Software Eng., vol. 17, no. 4, pp.313-319, April 1991.

[28] M. D. Smith and D. J. Robson, "A framework for testing object oriented programs,"J. of Object Oriented Programming, vol. 5, pp. 45-53, 1992.

[29] M. Z. Tsoukalas, J. W. Duran, S. C. Ntafos, "On some reliability estimation problems in random and partition testing," IEEE Trans. Software Eng., vol. 19, no. 7, pp.687-697, July 1993.

[30] C. D. Turner and D. J. Robson, "The state-based testing of object-oriented programs," In Proc. Conference on Software Maintenance(CSM-93), IEEE, pp. 302-310, Sept., 1993.

[31] S. Weerahandi and R. E. Hansmm, "Software quality measurement based on fault-detection data," IEEE Trans. Software Eng., vol. 20, no. 9, pp.665-674, Sept. 1994.

[32] E. J. Weyuker, "More experience with data flow testing," IEEE Trans. Software Eng., vol. 19, no. 9, pp.912-919, Sept. 1993.

Vita Auctoris

Peng (Paul) Luo was born on April 25, 1964 in Beijing, China. He graduated with the Bachelor of Engineering from Tsinghua University, Beijing, China in 1987. Master of Engineering from Institute of Semiconductors, Chinese Academy of Sciences, Beijing, China in 1990. After graduation He worked for about two years at Huaxia Silicon Valley Information System Ltd., Beijing, China, in various high-tech projects. He went to Tokyo Institute of Technology, Tokyo, Japan in 1992 as a Ph.D student. In 1993, he moved to Canada and studied in University of Windsor. During the course of his study, he worked for two years at Nortel, Ottawa, Canada, designing telecommunication software.